NAPALM Documentation

Release 1

David Barroso

May 12, 2018

Contents

YANG (RFC6020) is a data modelling language, it's a way of defining how data is supposed to look like. The napalm-yang library provides a framework to use models defined with YANG in the context of network management. It provides mechanisms to transform native data/config into YANG and vice versa.

CHAPTER 1

Documentation

1.1 Quickstart

1.1.1 Tutorial

You can take a look to the following tutorial to see what this is about and how to get started.

1.1.2 Installation

To install `napalm-yang` you can use pip as with any other driver:

```
pip install -U napalm-yang
```

1.2 YANG Basics

It's not really necessary to fully understand how YANG works to work with `napalm-yang` but understanding the language used by YANG can be beneficial to better understand the documentation and the benefits of it.

Here is a list or resources to start learning YANG:

- YANG for dummies

1.3 Supported Models

Below you can find all the YANG models supported and which profiles implements which ones. Note that all the implementations are not necessarily complete, in the next section you can find links to each individual profile so you can inspect them yourself.

1.4 Profiles

Profiles are responsible from mapping native data/configuration to a YANG model and vice versa. Below you can find links to all the profiles so you can inspect what each one does.

1.5 API

1.5.1 Models

Models are generated by `pyangbind` so it's better to check it's documentation for up to date information: http://pynms.io/pyangbind/generic_methods/

1.5.2 Utils

`napalm_yang.utils.`**`model_to_dict`**(*model*, *mode=''*, *show_defaults=False*)

Given a model, return a representation of the model in a dict.

This is mostly useful to have a quick visual represenation of the model.

Parameters

- **model** (*PybindBase*) – Model to transform.
- **mode** (*string*) – Whether to print config, state or all elements ("" for all)

Returns A dictionary representing the model.

Return type dict

Examples

```
>>> config = napalm_yang.base.Root()
>>>
>>> # Adding models to the object
>>> config.add_model(napalm_yang.models.openconfig_interfaces())
>>> config.add_model(napalm_yang.models.openconfig_vlan())
>>> # Printing the model in a human readable format
>>> pretty_print(napalm_yang.utils.model_to_dict(config))
>>> {
>>>     "openconfig-interfaces:interfaces [rw]": {
>>>         "interface [rw]": {
>>>             "config [rw]": {
>>>                 "description [rw]": "string",
>>>                 "enabled [rw]": "boolean",
>>>                 "mtu [rw]": "uint16",
>>>                 "name [rw]": "string",
>>>                 "type [rw]": "identityref"
>>>             },
>>>             "hold_time [rw]": {
>>>                 "config [rw]": {
>>>                     "down [rw]": "uint32",
>>>                     "up [rw]": "uint32"
    (trimmed for clarity)
```

`napalm_yang.utils.`**`diff`** (*f*, *s*)

Given two models, return the difference between them.

Parameters

- **f** (*Pybindbase*) – First element.
- **s** (*Pybindbase*) – Second element.

Returns A dictionary highlighting the differences.

Return type dict

Examples

```
>>> diff = napalm_yang.utils.diff(candidate, running)
>>> pretty_print(diff)
>>> {
>>>     "interfaces": {
>>>         "interface": {
>>>             "both": {
>>>                 "Port-Channel1": {
>>>                     "config": {
>>>                         "mtu": {
>>>                             "first": "0",
>>>                             "second": "9000"
>>>                         }
>>>                     }
>>>                 }
>>>             },
>>>             "first_only": [
>>>                 "Loopback0"
>>>             ],
>>>             "second_only": [
>>>                 "Loopback1"
>>>             ]
>>>         }
>>>     }
>>> }
```

1.5.3 Root

class `napalm_yang.base.`**`Root`**

Bases: `object`

This is a container you can use as root for your other models.

Examples

```
>>> config = napalm_yang.base.Root()
>>>
>>> # Adding models to the object
>>> config.add_model(napalm_yang.models.openconfig_interfaces())
>>> config.add_model(napalm_yang.models.openconfig_vlan())
```

add_model (*model*, *force=False*)
Add a model.

The model will be asssigned to a class attribute with the YANG name of the model.

Parameters

- **model** (*PybindBase*) – Model to add.
- **force** (*bool*) – If not set, verify the model is in SUPPORTED_MODELS

Examples

```
>>> import napalm_yang
>>> config = napalm_yang.base.Root()
>>> config.add_model(napalm_yang.models.openconfig_interfaces)
>>> config.interfaces
<pyangbind.lib.yangtypes.YANGBaseClass object at 0x10bef6680>
```

compliance_report (*validation_file='validate.yml'*)
Return a compliance report. Verify that the device complies with the given validation file and writes a compliance report file. See https://napalm.readthedocs.io/en/latest/validate.html.

elements ()

get (*filter=False*)
Returns a dictionary with the values of the model. Note that the values of the leafs are YANG classes.

Parameters **filter** (*bool*) – If set to `True`, show only values that have been set.

Returns A dictionary with the values of the model.

Return type dict

Example

```
>>> pretty_print(config.get(filter=True))
>>> {
>>>     "interfaces": {
>>>         "interface": {
>>>             "et1": {
>>>                 "config": {
>>>                     "description": "My description",
>>>                     "mtu": 1500
>>>                 },
>>>                 "name": "et1"
>>>             },
>>>             "et2": {
>>>                 "config": {
>>>                     "description": "Another description",
>>>                     "mtu": 9000
>>>                 },
>>>                 "name": "et2"
>>>             }
>>>         }
>>>     }
>>> }
```

load_dict (*data*, *overwrite=False*, *auto_load_model=True*)
Load a dictionary into the model.

Parameters

- **data** (*dict*) – Dictionary to loead
- **overwrite** (*bool*) – Whether the data present in the model should be overwritten by the data in the dict or not.
- **auto_load_model** (*bool*) – If set to true models will be loaded as they are needed

Examples

```
>>> vlans_dict = {
>>>     "vlans": { "vlan": { 100: {
>>>                             "config": {
>>>                                 "vlan_id": 100, "name": "production"}},
>>>                          200: {
>>>                             "config": {
>>>                                 "vlan_id": 200, "name": "dev"}}}}}
>>> config.load_dict(vlans_dict)
>>> print(config.vlans.vlan.keys())
... [200, 100]
>>> print(100, config.vlans.vlan[100].config.name)
... (100, u'production')
>>> print(200, config.vlans.vlan[200].config.name)
... (200, u'dev')
```

parse_config (*device=None*, *profile=None*, *native=None*, *attrs=None*)
Parse native configuration and load it into the corresponding models. Only models that have been added to the root object will be parsed.

If `native` is passed to the method that's what we will parse, otherwise, we will use the `device` to retrieve it.

Parameters

- **device** (*NetworkDriver*) – Device to load the configuration from.
- **profile** (*list*) – Profiles that the device supports. If no `profile` is passed it will be read from `device`.
- **native** (*list of strings*) – Native configuration to parse.

Examples

```
>>> # Load from device
>>> running_config = napalm_yang.base.Root()
>>> running_config.add_model(napalm_yang.models.openconfig_interfaces)
>>> running_config.parse_config(device=d)
```

```
>>> # Load from file
>>> with open("junos.config", "r") as f:
>>>     config = f.read()
>>>
>>> running_config = napalm_yang.base.Root()
>>> running_config.add_model(napalm_yang.models.openconfig_interfaces)
>>> running_config.parse_config(native=[config], profile="junos")
```

parse_state (*device=None*, *profile=None*, *native=None*, *attrs=None*)

Parse native state and load it into the corresponding models. Only models that have been added to the root object will be parsed.

If `native` is passed to the method that's what we will parse, otherwise, we will use the `device` to retrieve it.

Parameters

- **device** (*NetworkDriver*) – Device to load the configuration from.
- **profile** (*list*) – Profiles that the device supports. If no `profile` is passed it will be read from `device`.
- **native** (*list string*) – Native output to parse.

Examples

```
>>> # Load from device
>>> state = napalm_yang.base.Root()
>>> state.add_model(napalm_yang.models.openconfig_interfaces)
>>> state.parse_config(device=d)
```

```
>>> # Load from file
>>> with open("junos.state", "r") as f:
>>>     state_data = f.read()
>>>
>>> state = napalm_yang.base.Root()
>>> state.add_model(napalm_yang.models.openconfig_interfaces)
>>> state.parse_config(native=[state_data], profile="junos")
```

to_dict (*filter=True*)

Returns a dictionary with the values of the model. Note that the values of the leafs are evaluated to python types.

Parameters **filter** (*bool*) – If set to `True`, show only values that have been set.

Returns A dictionary with the values of the model.

Return type dict

Example

```
>>> pretty_print(config.to_dict(filter=True))
>>> {
>>>     "interfaces": {
>>>         "interface": {
>>>             "et1": {
>>>                 "config": {
>>>                     "description": "My description",
>>>                     "mtu": 1500
>>>                 },
>>>                 "name": "et1"
>>>                 },
>>>                 "et2": {
>>>                     "config": {
>>>                         "description": "Another description",
>>>                         "mtu": 9000
>>>                     },
>>>                     "name": "et2"
>>>                 }
>>>             }
>>>         }
>>> }
```

translate_config(*profile*, *merge=None*, *replace=None*)

Translate the object to native configuration.

In this context, merge and replace means the following:

- **Merge** - Elements that exist in both `self` and `merge` will use by default the values in `merge` unless `self` specifies a new one. Elements that exist only in `self` will be translated as they are and elements present only in `merge` will be removed.
- **Replace** - All the elements in `replace` will either be removed or replaced by elements in `self`.

You can specify one of `merge`, `replace` or none of them. If none of them are set we will just translate configuration.

Parameters

- **profile** (*list*) – Which profiles to use.
- **merge** (`Root`) – Object we want to merge with.
- **replace** (`Root`) – Object we want to replace.

1.6 Developers Guide

WIP. Information here is a bit chaotic, sorry about that.

1.6.1 Profiles

In order to correctly map YANG objects to native configuration and vice versa, `napalm-yang` uses the concept of **profiles**. Profiles, identify the type of device you are dealing with, which can vary depending on the OS, version and/or platform you are using.

If you are using a napalm driver and have access to your device, you will have access to the `profile` property which you can pass to any function that requires to know the profile. If you are not using a napalm driver or don't have access to the device, a profile is just a list of strings so you can just specify it directly. For example:

```
# Without access to the device
model.parse_config(profile=["junos"], config=my_configuration)

# With access
with driver(hostname, username, password) as d:
    model.parse_config(device=d)

# With access but overriding profile
```

```
with driver(hostname, username, password) as d:
    model.parse_config(device=d, profile=["junos13", "junos"])
```

Note: As you noticed a device may have multiple profiles. When that happens, each model that is parsed will loop through the profiles from left to right and use the first profile that implements that model (note that a YANG model is often comprised of multiple modules). This is useful as there might be small variances between different systems but not enough to justify reimplementing everything.

You can find the profiles here but what exactly is a profile? A profile is a bunch of YAML files that follows the structure of a YANG model and describes two things:

1. How to parse native configuration/state and map it into a model.
2. How to translate a model and map it into native configuration.

For example, here you can see how to map native configuration from an EOS device into the `openconfig-interface` model and here how to map the model to native configuration.

As you can see it's not extremely difficult to understand what they are doing, in the next section we will learn how to write our own profiles.

1.6.2 Writing Profiles

As it's been already mentioned, a profile consists of a bunch of YAML files that describe how to map native configuration and how to translate an object into native configuration. In order to read native configuration we will use **parsers**. To translate a YANG model into native configuration we will use **translators**.

Both parsers and translators follow three basic rules:

1. One directory per module.
2. One file per model.
3. Exact same representation of the model inside the file:

For example:

```
$ tree napalm_yang/mappings/eos/parsers/config
napalm_yang/mappings/eos/parsers/config
├── napalm-if-ip
│   └── secondary.yaml
├── openconfig-if-ip
│   └── ipv4.yaml
├── openconfig-interfaces
│   └── interfaces.yaml
└── openconfig-vlan
    ├── routed-vlan.yaml
    └── vlan.yaml

4 directories, 5 files
$ cat napalm_yang/mappings/eos/parsers/config/openconfig-vlan/vlan.yaml
---
metadata:
    (trimmed for brevity)

vlan:
    (trimmed for brevity)
    config:
        (trimmed for brevity)
        vlan_id:
            (trimmed for brevity)
```

If we check the content of the file `vlan.yaml` we can clearly see two parts:

- **metadata** - This part specifies what parser or translator we want to use. There are several options available depending on the type of data we are parsing from or translating to. Additionally, we need to provide some options that the parser/translator might need. For example:

  ```
  metadata:
      processor: XMLParser
      execute:
          - method: _rpc
            args: []
            kwargs:
                get: "<get-configuration/>"
  ```

In this case we are using the `XMLParser` parser. In order to get the data we need from the device we have to call the method `_rpc` with the `args` and `kwargs` parameters. This is, by the way, an RPC call for a junos device.

Note: If a model is called by other models, like openconfig-if-ethernet is called by openconfig-interfaces, there is no need to specify the execute section in the metadata, if the commands will be the same. Since the first model executes the commands, the data from the commands will be shared with the models that the first model calls. While it will work it to specify the same commands in the metadata section for the second model, it could cause the commands to be executed multiple times.

- **vlan** - This is the part that follows the model specification. In this case is `vlan` but in others it might be `interfaces`, `addressess` or something else, this will be model dependent but it's basically whatever it's not `metadata`. This part will follow the model specification and add rules on each attribute to tell the parser/translator what needs to be done. For example:

  ```
  vlan:
      _process: unnecessary
      config:
          _process: unnecessary
          vlan_id:
              _process:
                  - mode: xpath
                    xpath: "vlan-id"
                    from: "{{ parse_bookmarks['parent'] }}"
  ```

We have to specify the `_process` attribute at each step, which can either be `unnecessary`, ‘‘ not_implemented‘‘ or a list of rules:

- `not_implemented` means that we haven't added support to that field. In addition it will stop parsing that branch of the tree.
- `unnecessary` means that we don't need that field. This is common in containers as you usually don't need to process them at all.
- `list` of rules. See *Parsers* and *Translators*.

Something else worth noting is that each rule inside `_process` is evaluated as a `jinja2` template so you can do variable substitutions, evaluations, etc...

1.6.3 Parsers

Parsers are responsible for mapping native configuration/data to a YANG model.

Processing data

The first thing you have to know is what type of data you are dealing with and then select the appropiate parser. Each one initializes the data and makes it available in similar ways but you have to be aware of the particularities of each one.

You can select the parser with the metadata field:

```
---
metadata:
    processor: TextTree
    execute:
        - method: cli
          kwargs:
              commands: ["show running-config all"]
```

That block not only specifies which parser to use but how to retrieve the data necessary for the parser to operate for that particular model.

Available processors are:

JSONParser

TBD

XMLParser

TBD

TextParser

TBD

Rule Directives

Keys

When a list is traversed you will always have available a key with name `$(attribute)_key`. In addition, you will have `parent_key` as the key of the immediate parent object. Example.

Bookmarks

Bookmarks are points of interest in the configuration. Usually, you will be gathering blocks of configurations and parsing on those as you progress. However, sometimes the data you need is somewhere else. For those cases you can use the bookmarks within the `from` field to pick the correct block of configuration.

Bookmarks are created automatically according to these rules:

- At the begining of each model a bookmark of name `root_$first_attribute_name` will point to the list of data that the parser requires. Example
- When a container is traversed, a bookmark will be created with name `$attribure_name`
- When a list is traveresed, each element of the list will have its own bookmark with name `$attribute_name.$key`.

`extra_vars`

The `regexp` directive lets you capture any arbitary amount of information you want. All captured groups will be avaible to you inside the `extra_vars.$attribute` (`$attribute` is the attribute where the additional information was captured). Example.

Examples

Examples - Lists

Parsing interfaces in industry standard CLIs (simple case)

When TexTree parses industry standard CLIs it will generate a dictionary similar to:

```
interface:
    Fa1:
        '#standalone': true
        '#text': no shutdown
        'no':
          '#text': shutdown
          shutdown:
            '#standalone': true
        # other data relevant to the interface
    Fa2:
        '#standalone': true
        '#text': shutdown
        shutdown:
          '#standalone': true
        # other data relevant to the interface
```

This means that to parse the interfaces we only have to advance to the `interface` key and map the keys to the YANG model key and get the block for further processing.

Original data

```
interface Port-Channel1
   shutdown
!
interface Port-Channel1.1
   shutdown
!
interface Ethernet1
   shutdown
!
interface Ethernet2
   no shutdown
```

```
!
interface Ethernet2.1
   no shutdown
!
interface Ethernet2.2
   no shutdown
!
interface Loopback1
   no shutdown
!
interface Management1
   no shutdown
!
```

Parser rule

```
- from: root_interfaces.0
  path: interface
  regexp: ^(?P<value>(\w|-)*\d+(\/\d+)*)$
```

- `regexp` is useful to filter out data that we don't want to process. For example, in the example above we are basically filtering subinterfaces as they will be processed later. Note that the regular expression has to capture a `value`.
- `path` is simply telling the parser that the data is looking for is inside the `interface` key.
- `from` is just telling the parser where to get the data from. This is the first element processed by the profile so there is no information that can be inferred yet.

Result

Note that `extra_vars` will be populated with anything you capture with the regular expression. This might be handier when parsing more complex keys like ip addresses which might include the prefix length.

Note as well that we didn't get any subinterface thanks to `regexp`.

Example 1

```
extra_vars: {}
keys: {}
```

Parsing subinterfaces in industry standard CLIs (variables)

When we were parsing interfaces we skipped the subinterfaces. In order to pass subinterfaces we can leverage on the `interface_key` to build a dynamic regular expression.

Original data

```
interface Port-Channel1
   shutdown
!
interface Port-Channel1.1
   shutdown
!
interface Ethernet1
   shutdown
!
interface Ethernet2
   no shutdown
!
interface Ethernet2.1
   no shutdown
!
interface Ethernet2.2
   no shutdown
!
interface Loopback1
   no shutdown
!
interface Management1
   no shutdown
!
```

Parser rule

```
- path: interface
  regexp: '{{interface_key}}\.(?P<value>\d+)'
```

Because we are parsing a *subinterface* which is a child of an *interface*, all the keys and extra_vars that we previously collected in the current interface will be available. We will use `{{ interface_key }}` in our regular expression to match only our current parent interface.

Result

Note that thanks to the variable used in the regular expression we are only capturing the relevant subinterface for the current interface. In the second case it turns out there are no subinterfaces.

Example 1

```
extra_vars: {}
keys:
  interface_key: Ethernet2
```

Example 2

```
extra_vars: {}
keys:
  interface_key: Loopback1
```

Parsing IP addresses in EOS (extracting extra information from a key)

IP addresses in EOS contain two pieces of information; the address and it's prefix-length. You can use `regexp` to select the relevant part for the key and any additional information you may need.

Original data

```
ip address 192.168.1.1/24
ip address 192.168.2.1/24 secondary
ip address 172.20.0.1/24 secondary
```

Parser rule

```
- path: ip.address
  regexp: (?P<value>(?P<ip>.*))\/(?P<prefix>\d+)
```

The regular expression is doing two things; use the `<value>` to capture which part should be used for the key and then capture as well all the useful information so we have it available for later use in the `extra_vars` field.

Result

Note that `extra_vars` is populated with the information we captured with `regexp`..

Example 1

```
extra_vars: {}
keys: {}
```

Parsing IP addresses in IOS (flattening dictionaries)

Sometimes the information is unnecessarily nested. This is the case for the ip address configuration in IOS. Let's see how that data might look like after processing it with the TextParser:

```
ip:
    address:
        192.168.2.1:
            255.255.255.0:
                secondary:
                    "#standalone": true
        192.168.1.1": {
            255.255.255.0:
                "#standalone": true
        172.20.0.1:
            255.255.255.0:
            secondary":
                "#standalone": true
```

Luckily, we can solve this issue with the `path` resolver.

Original data

```
ip address 192.168.1.1 255.255.255.0
ip address 192.168.2.1 255.255.255.0 secondary
ip address 172.20.0.1 255.255.255.0 secondary
```

Parser rule

```
- key: prefix
  path: ip.address.?prefix.?mask
  regexp: ^(?P<value>\d+\.\d+\.\d+\.\d+)
```

We specify a `regexp` here to make sure we don't parse lines like `ip address dhcp`.

When path contains `?identifier` what it actually does is flatten that key and assign the value of that key to a new key named `identifier`. For example, with the nested structure and the path we have right now we would get the following:

```
- prefix: 192.168.1.1
  mask: 255.255.255.0
  '#standalone': true
- prefix: 192.168.2.1
  mask: 255.255.255.0
  secondary:
    '#standalone': true
- prefix: 172.20.0.1
  mask: 255.255.255.0
  prefix: 172.20.0.1
  secondary:
    '#standalone': true
```

Result

Example 1

```
extra_vars: {}
keys: {}
```

Parse BGP neighbors in Junos (nested lists)

XML often consists of lists of lists of lists which sometimes makes it challenging to nest things in a sane manner. Hopefully, the `path` can solve this issue as well.

Original data

```
<some_configuration_block>
    <group>
        <name>my_peers</name>
        <neighbor>
```

```
            <name>192.168.100.2</name>
            <description>adsasd</description>
            <peer-as>65100</peer-as>
        </neighbor>
        <neighbor>
            <name>192.168.100.3</name>
            <peer-as>65100</peer-as>
        </neighbor>
    </group>
    <group>
        <name>my_other_peers</name>
        <neighbor>
            <name>172.20.0.1</name>
            <peer-as>65200</peer-as>
        </neighbor>
    </group>
</some_configuration_block>
```

Parser rule

```
- key: ip
  path: group.?peer_group:name.neighbor.?ip:name
```

Note that this time the path contains a couple of `?identifier:field`. That pattern is used to flatten lists and what it does is assign the contents of that sublist to the parent object and also assign the value of `field` to a new `key` called `identifier`. For example, the XML above will be converted to the following structure:

```
- name:
    '#text': my_peers
  peer-as:
    '#text': 65100
  neighbor: 192.168.100.3
  peer_group: my_peers
- name:
    '#text': my_peers
  description:
    '#text': adsasd
  peer-as:
    '#text': 65100
  neighbor: 192.168.100.2
  peer_group: my_peers
- name:
    '#text': my_other_peers
  peer-as:
    '#text': 65200
  neighbor: 172.20.0.1
  peer_group: my_other_peers
```

Result

Example 1

```
extra_vars: {}
keys: {}
```

Parsing protocols (down the rabbit hole)

Some parsing might require more complex rules. In this example we can see how to combine multiple rules ran under different circumstances.

Original data

```
ip route 10.0.0.0/24 192.168.0.2 10 tag 0
ip route vrf devel 10.0.0.0/24 192.168.2.2 1 tag 0
!
router bgp 65001
   router-id 1.1.1.1
   address-family ipv4
      default neighbor 192.168.0.200 activate
   !
   address-family ipv6
      default neighbor 192.168.0.200 activate
   vrf devel
      router-id 3.3.3.3
!
router pim sparse-mode
   vrf devel
      ip pim log-neighbor-changes
!
```

Parser rule

```
- key: '{{ protocol }} {{ protocol }}'
  path: router.?protocol.?process_id
  regexp: (?P<value>bgp bgp)
  when: '{{ network_instance_key == ''global'' }}'
- from: root_network-instances.0
  key: '{{ protocol }} {{ protocol }}'
  path: router.?protocol.?process_id.vrf.{{ network_instance_key }}
  regexp: (?P<value>bgp bgp)
  when: '{{ network_instance_key != ''global'' }}'
- from: root_network-instances.0
  key: '{{ ''static static'' }}'
  path: ip.route
```

When multiple rules are specified all of them will be executed and the results will be concatenated. You can combine this technique with `when` to specify how to parse the data under different circumstances (see rules `#1` and `#2`) or just to add more ways of parsing data (see rule `#3`)

Note also that we are also dynamically building the `key` to follow the format that the YANG model requires, which in this case is as simple (and weird) as just specifying a name for our protocol (which in our case will be the same as the protocool).

It also worth noting that we are using a regular expression to match only on `BGP`. We are doing that to avoid processing protocols that we are not (yet) supporting in this profile.

Result

The results below might look intimidating but it's basically the relevant configuration for BGP and for the static routes for the current `network_instance`.

Example 1

```
extra_vars: {}
keys:
  network_instance_key: global
```

Example 2

```
extra_vars: {}
keys:
  network_instance_key: devel
```

Parsing json interfaces IOS-XE (jsonrpc)

IOS-XE groups interfaces by type.

Original data

```
{
  "Cisco-IOS-XE-native:interface": {
    "GigabitEthernet": [
      {
        "name": "1",
        "ip": {
          "address": {
            "dhcp": {
            }
          }
        },
        "mop": {
          "enabled": false
        },
        "Cisco-IOS-XE-ethernet:negotiation": {
          "auto": true
        }
      },
      {
        "name": "2",
        "description": "GbE 2",
        "ip": {
          "no-address": {
            "address": false
          }
        },
        "mop": {
          "enabled": false
        },
        "Cisco-IOS-XE-ethernet:negotiation": {
          "auto": true
        }
      },
      {
        "name": "2.10",
        "description": "GbE 2.10",
        "encapsulation": {
          "dot1Q": {
            "vlan-id": 10
          }
        },
        "vrf": {
          "forwarding": "internal"
        },
        "ip": {
          "address": {
            "primary": {
              "address": "172.16.10.1",
              "mask": "255.255.255.0"
            }
          }
        }
      }
      ],
    "Loopback": [
      {
        "name": 0,
        "description": "Loopback Zero",
        "ip": {
          "address": {
            "primary": {
              "address": "100.64.0.1",
              "mask": "255.255.255.255"
            }
          }
        },
        "ipv6": {
          "address": {
            "prefix-list": [
              {
                "prefix": "2001:DB8::1/64"
              }
            ]
          }
        }
      },
      {
        "name": 1,
        "description": "Loopback One",
        "vrf": {
          "forwarding": "mgmt"
        },
        "ip": {
          "no-address": {
            "address": false
          }
        }
      }
    ]
  }
}
```

Parser rule

```
- from: root_interfaces.0
  key: '{{ type }}{{ name }}'
  path: Cisco-IOS-XE-native:interface.?type
  regexp: ^(?P<value>(\w|-)*\d+(\/\d+)*)$
```

Result

Example 1

```
extra_vars: {}
keys: {}
```

Parsing ntp peers Junos

Junos groups ntp servers and peers by type and then lists them

Original data

```
<configuration>
        <system>
            <ntp>
                <peer>
                    <name>172.17.19.1</name>
                </peer>
                <server>
                    <name>172.17.19.2</name>
                    <name>172.17.19.3</name>
                </server>
            </ntp>
        </system>
</configuration>
```

Parser rule

```
- key: '#text'
  path: configuration.system.ntp.?type.name
```

Result

Example 1

```
extra_vars: {}
keys: {}
```

Examples - leaf

Parsing metric style for junos ISIS level

Junos sometimes indicates the value of something not by using an element with a specific value but by the presence of it. For instance, the metric style of an ISIS level can either be NARROW or WIDE but instead of indicating explicitly junos will do it with the presence or not of the element `</wide-metrics-only>`.

Original data

```
<configuration>
    <isis>
        <level>
            <name>1</name>
            <wide-metrics-only/>
        </level>
        <level>
            <name>2</name>
            <preference>50</preference>
        </level>
        <interface>
            <name>ge-0/0/0.0</name>
            <point-to-point/>
        </interface>
        <interface>
            <name>ge-0/0/1.0</name>
            <point-to-point/>
        </interface>
        <interface>
            <name>ge-0/0/2.0</name>
            <point-to-point/>
        </interface>
        <interface>
            <name>ge-0/0/3.0</name>
            <point-to-point/>
        </interface>
        <interface>
            <name>lo0.0</name>
        </interface>
    </isis>
</configuration>
```

Parser rule

```
- default: NARROW_METRIC
  path: wide-metrics-only
  post: WIDE_METRIC
  present: true
```

Result

Example 1

```
extra_vars: {}
keys: {}
```

Parse EOS privilege:role

The role in eos is the combination of the privilege level and the role itself. In this example we should how to use a regular expression to capture data and postprocess it to set the correct role.

Original data

```
username test1 privilege 1 nopassword
username test2 privilege 1 secret sha512 $6$WL/ibPpzPJ/C7c/E$.
↪bVF08dYhlNp0rxER0P3SNdsA2wUtK2Ru1YuKkRRZQGl609DA1JvX.dSFgKXaq.
↪LWjDRlZoHudfk7hamod0Th/
username test3 privilege 15 role network-operator secret sha512 $6$Vd6.7k2FybfsTKKp$S.
↪AHfdwicaWEoA41sPd6ZXOOdruJMrKJh70WNfiX/eZKH1oYBtFz9VbrPlYNDkhM/pi54gcYKH2hviy/xrUav.
username test4 privilege 1 secret 5 $1$NKhJ$PUfYNtJF2tIneEBZztchy.
username test5 privilege 15 secret sha512 $6$d3fdbbZBrhplknVB$FILKNelLURwd/
↪xT74ktjxJ4XP1vTfJ53H7OWJHgAqeuY/lF3BDyP3SWpH/MeBRnl7lLi8hU2oy6hkbnB7jvtA.
username test6 privilege 1 role network-admin secret sha512 $6$zaalm5RTm6/26XVS$I/
↪f3kmOqfvTbjwjzepCe1O9eYfPJRdUrRLe9NoMsbgNz9T48nj0AlOsm2LmoFp6aI5B6Q/xlseJdNrTL/jiXH0
username test7 privilege 15 role network-admin secret 5 $1$NKhJ$PUfYNtJF2tIneEBZztchy.
```

Parser rule

```
- path: privilege
  post: '{{ extra_vars.privilege }}:{{ extra_vars.role }}'
  regexp: (?P<value>(?P<privilege>\d+).*role (?P<role>\S+)) secret
- path: privilege
  post: '{{ value }}:network-operator'
  regexp: (?P<value>\d+)
```

Result

Example 1

```
extra_vars: {}
keys: {}
```

1.6.4 Translators

Translators are responsible for transforming a model into native configuration.

Special actions

Most actions depend on the parser you are using, however, some are common to all of them:

unnecessary

This makes the parser skip the field and continue processing the tree.

not_implemented

This makes the parser stop processing the tree underneath this value. For example:

```
field_1:
    process: unnecessary
field_2:
    process: not_implemented
    subfield_1:
        process: ...
    subfield_2:
        process: ...
field_3:
    ...
```

The `not_implemented` action will stop the parser from processing `subfield_1` and `subfield_2` and move directly onto `field_3`.

gate

Works like `not_implemented` but accepts a condition. For example:

```
protocols:
    protocol:
        bgp:
            _process:
              - mode: gate
                when: "{{ protocol_key != 'bgp bgp' }}"
            global:
                ...
```

The snippet above will only process the `bgp` subtree if the condition is **not** met.

Special fields

When translating an object, some fields might depend on the translator you are using but some will available regardless. Some may be even be mandatory.

mode

- **mandatory**: yes
- **description**: which parsing/translation action to use for this particular field
- **example**: translate description attribute of an interface to native configuration:

```
description:
    _process:
        - mode: element
          value: "    description {{ model }}\n"
          negate: "    default description"
```

when

- **mandatory**: no
- **description**: the evaluation of this field will determine if the action is executed or skipped. This action is probably not very useful when parsing but it's available if you need it.
- **example**: configure `switchport` on IOS devices only if the interface is not a loopback or a management interface:

```
ipv4:
    _process: unnecessary
    config:
        _process: unnecessary
        enabled:
            _process:
                - mode: element
                  value: "    no switchport\n"
                  negate: "    switchport\n"
                  in: "interface.{{ interface_key }}"
                  when: "{{ model and interface_key[0:4] not in ['mana', 'loop'] }
↪}"
```

in

- **mandatory**: no
- **description**: where to add the configuration. Sometimes the configuration might have to be installed on a different object from the one you are parsing. For example, when configuring a tagged subinterface on junos you will have to add also a `vlan-tagging` option on the parent interface. On `IOS/EOS`, when configuring interfaces, you have to also add the configuration in the root of the configuration and not as a child of the parent interface:

```
vlan:
    _process: unnecessary
    config:
        _process: unnecessary
        vlan_id:
            _process:
                - mode: element
                  element: "vlan-tagging"
                  in: "interface.{{ interface_key }}" # <--- add element to␣
↪parent interface
                  when: "{{ model > 0 }}"
                  value: null
                - mode: element
                  element: "vlan-id"
                  when: "{{ model > 0 }}"

(...)
subinterface:
    _process:
        mode: container
        key_value: "interface {{ interface_key}}.{{ subinterface_key }}\n"
        negate: "no interface {{ interface_key}}.{{ subinterface_key }}\n"
        in: "interfaces"                            # <--- add element to root of␣
↪configuration
```

Note: This field follows the same logic as the *Bookmarks* special field.

continue_negating

- **mandatory**: no
- **description**: this option, when added to a container, will make the framework to also negate children.
- **example**: we can use as an example the "network-instances" model. In the model, BGP is inside the `network-instance` container, however, in EOS and other platforms that BGP configuration is decoupled from the VRF, so in order to tell the framework to delete also the direct children you will have to use this option. For example:

```
network-instance:
    _process:
        - mode: container
          key_value: "vrf definition {{ network_instance_key }}\n"
          negate: "no vrf definition {{ network_instance_key }}\n"
          continue_negating: true
          end: "    exit\n"
          when: "{{ network_instance_key != 'global' }}"
    ...
    protocols:
        _process: unnecessary
        protocol:
            _process:
              - mode: container
                key_value: "router bgp {{ model.bgp.global_.config.as_ }}\n  vrf {
↪{ network_instance_key}}\n"
                negate: "router bgp {{ model.bgp.global_.config.as_ }}\n  no vrf {
↪{ network_instance_key}}\n"
```

```
                end: "    exit\n"
                when: "{{ protocol_key == 'bgp bgp' and network_instance_key !=
↪'global' }}"
                replace: false
                in: "network-instances"
```

The example above will generate:

```
no vrf definition blah
router bgp ASN
   no vrf blah
```

Without `continue_negating` it would just generate:

```
no vrf definition blah
```

Special variables

keys

See *Keys*.

model

This is the current model/attribute being translated. You have the entire object at your disposal, not only it's value so you can do things like:

```
vlan_id:
    _process:
        - mode: element
          value: "    encapsulation dot1q vlan {{ model }}\n"
```

Or:

```
config:
    _process: unnecessary
    ip:
        _process: unnecessary
    prefix_length:
        _process:
            - mode: element
              value: "    ip address {{ model._parent.ip }}/{{ model }} {{ 'secondary
↪' if model._parent.secondary else '' }}\n"
              negate: "    default ip address {{ model._parent.ip }}/{{ model }}\n"
```

1.6.5 XMLTranslator

XMLTranslator is responsible for translating a model into XML configuration.

Metadata

- **xml_root** - Set this value on the root of the model to instantiate the XML object.

For example:

```
---
metadata:
    processor: XMLTranslator
    xml_root: configuration
```

This will instantiate the XML object `<configuration/>`.

Container - container

Creates a container.

Arguments:

- **container** (mandatory) - Container name.
- **replace** (optional) - True/Flase, depending Whether this element has to be replaced in case of merge/replace or it's not necessary (remember XML is hierarchical, which means you can unset things directly in the root).

Example:

Create the `interfaces` container:

```
_process:
  . mode: container
    container: interfaces
    replace: true
```

List - container

For each element of the list, create a container.

Arguments:

- **container** (mandatory) - Name of container to create.
- **key_element** (mandatory) - Lists require a key element, this is the name of the element.
- **key_value** (mandatory) - Key element value.

Example:

Create interfaces:

```
interface:
    _process:
      . mode: container
        container: interface
        key_element: name
        key_value: "{{ interface_key }}"
```

This will result elements such as:

```
<interface>
  <name>ge-0/0/0</name>
</interface>
<interface>
  <name>lo0</name>
</interface>
```

Leaf - element

Adds an element to a container.

Arguments:

- **element** (mandatory): Element name.
- **value** (optional): Override value. Default is value of the object.

Example 1:

Configure description:

```
description:
    _process:
        - mode: element
          element: description
```

Example 2:

Enable or disable an interface:

```
enabled:
    _process:
        - mode: element
          element: "disable"
          when: "{{ not model }}"
          value: null
```

We override the value and set it to `null` because to disable we just have to create the element, we don't have to set any value.

Example 3:

Configure an IP address borrowing values from other fields:

```
config:
    _process: unnecessary
    ip:
        _process: unnecessary
    prefix_length:
        _process:
            - mode: element
              element: name
              value: "{{ model._parent.ip }}/{{ model }}"
              when: "{{ model }}"
```

1.6.6 TextTranslator

TextTranslator is responsible of translating a model into text configuration.

Metadata

- **root** - Set to true if this is the root of the model.

List - container

Create/Removes each element of the list.

Arguments:

- **key_value** (mandatory): How to create the element.
- **negate** (mandatory): How to eliminate/default the element.
- **replace** (optional): Whether the element has to be defaulted or not during the replace operation.
- **end** (optional): Closing command to signal end of element

Example 1:

Create/Default interfaces:

```
interfaces:
    _process: unnecessary
    interface:
        _process:
          . mode: container
            key_value: "interface {{ interface_key }}\n"
            negate: "{{ 'no' if interface_key[0:4] in ['Port', 'Loop'] else
↪'default' }} interface {{ interface_key }}\n"
            end: "    exit\n"
```

Example 2:

Configure IP addresses. As the parent interface is defaulted already, don't do it again:

```
address:
    _process:
      . mode: container
        key_value: "    ip address {{ model.config.ip }} {{ model.config.
↪prefix_length|cidr_to_netmask }}{{ ' secondary' if model.config.secondary␣
↪else '' }}\n"
        negate: "    default ip address {{ model.config.ip }} {{ model.
↪config.prefix_length|cidr_to_netmask }}{{ ' secondary' if model.config.
↪secondary else '' }}\n"
        replace: false
```

Leaf - element

Configures an attribute.

Arguments:

- **value** (mandatory): How to configure the attribute
- **negate** (mandatory): How to default the attribute

Example 1:

Configure description:

```
description:
    _process:
        - mode: element
          value: "    description {{ model }}\n"
          negate: "    default description"
```

Example 2:

Configure an IP address borrowing values from other fields:

```
address:
    _process: unnecessary
    config:
        _process: unnecessary
        ip:
            _process: unnecessary
        prefix_length:
            _process:
                - mode: element
                  value: "    ip address {{ model._parent.ip }}/{{ model }} {
↪{ 'secondary' if model._parent.secondary else '' }}\n"
                  negate: "    default ip address {{ model._parent.ip }}/{{␣
↪model }} {{ 'secondary' if model._parent.secondary else '' }}\n"
```

1.6.7 Jinja2 Filters

IP address

napalm_yang.jinja_filters.ip_filters.**cidr_to_netmask**(*value*, **args*, ***kwargs*)
Converts a CIDR prefix-length to a network mask.

Examples

```
>>> "{{ '24'|cidr_to_netmask }}" -> "255.255.255.0"
```

napalm_yang.jinja_filters.ip_filters.**netmask_to_cidr**(*value*, **args*, ***kwargs*)
Converts a network mask to it's CIDR value.

Examples

```
>>> "{{ '255.255.255.0'|netmask_to_cidr }}" -> "24"
```

napalm_yang.jinja_filters.ip_filters.**normalize_address**(*value*, **args*, ***kwargs*)
Converts an IPv4 or IPv6 address writen in various formats to a standard textual representation.

This filter works only on addresses without network mask. Use normalize_prefix to normalize networks.

Examples

```
>>> "{{ '192.168.0.1'|normalize_address }}" -> "192.168.0.1"
>>> "{{ '192.168.1'|normalize_address }}" -> "192.168.0.1"
>>> "{{ '2001:DB8:0:0:1:0:0:1'|normalize_address }}" -> "2001:db8::1:0:0:1"
```

napalm_yang.jinja_filters.ip_filters.**normalize_prefix**(*value*, **args*, ***kwargs*)
Converts an IPv4 or IPv6 prefix writen in various formats to its CIDR representation.

This filter works only on prefixes. Use normalize_address if you wish to normalize an address without a network mask.

Examples

```
>>> "{{ '192.168.0.0 255.255.255.0'|normalize_prefix }}" -> "192.168.0.0/24"
>>> "{{ '192.168/255.255.255.0'|normalize_prefix }}" -> "192.168.0.0/24"
>>> "{{ '2001:DB8:0:0:1:0:0:1/64'|normalize_prefix }}" -> "2001:db8::1:0:0:1/64"
```

`napalm_yang.jinja_filters.ip_filters.`**`prefix_to_addrmask`**(*value*, **args*, ***kwargs*)

Converts a CIDR formatted prefix into an address netmask representation. Argument sep specifies the separator between the address and netmask parts. By default it's a single space.

Examples

```
>>> "{{ '192.168.0.1/24|prefix_to_addrmask }}" -> "192.168.0.1 255.255.255.0"
>>> "{{ '192.168.0.1/24|prefix_to_addrmask('/') }}" -> "192.168.0.1/255.255.255.0"
```

1.6.8 FAQ

Some YAML files are insanely large. Can I break them down into multiple files?

Yes, you can with the `!include relative/path/to/file.yaml` directive. For example:

```
# ./main.yaml
my_key:
    blah: asdasdasd
    bleh: !include includes/bleh.yaml

# ./includes/bleh.yaml
qwe: 1
asd: 2
```

Will result in the final object:

```
my_key:
    blah: asdasdasd
    bleh:
        qwe: 1
        asd: 2
```

Python Module Index

Index